Dynamic Domains in Data Production Planning

Keith Golden **Wanlin Pang***
NASA Ames Research Center
Moffett Field, CA 94035
{kgolden, wpang}@email.arc.nasa.gov

Abstract

This paper discusses a planner-based approach to automating data production tasks, such as producing fire forecasts from satellite imagery and weather station data. Since the set of available data products is large, dynamic and mostly unknown, planning techniques developed for closed worlds are unsuitable. We discuss a number of techniques we have developed to cope with data production domains, including a novel constraint propagation algorithm based on planning graphs and a constraint-based approach to interleaved planning, sensing and execution.

1 Introduction

Petabytes of remote sensing data are now available from Earth-observing satellites to help measure, understand and forecast changes in the Earth system, but using these data effectively can be surprisingly hard. The volume and variety of data files and formats are daunting. Simple data management activities, such as locating and transferring files, changing file formats, gridding point data, and scaling and reprojecting gridded data, can consume far more personnel time and resources than the actual data analysis. We address this problem by developing a planner-based agent for data production, called IMAGEbot [Golden *et al.*, 2003], that takes data product requests as high-level goals and executes the commands needed to produce the requested data products.

The data production problem consists of converting an initial set of low-level data products into higher-level data products that can be used for science or decision support. The data products we are concerned with are geospatial data measuring specific *variables* of the Earth system, such as precipitation, vegetation productivity and fire risk, but our approach is also applicable to other types of data. Higher-level data products may be transformed versions of lower-level data products, or they may be entirely new products providing estimates or predictions of unknown Earth system variables, such as soil moisture, based on known variables, such as precipitation. These variables are estimated by running one or more computational *models*, such as simulation codes. The models can be precisely characterized in terms of their input and output requirements, which makes them straightforward to represent in an AI planning system. However, there are significant differences between the data production problem and more traditional planning domains, calling for different techniques.

Notable features of data processing domains include large dynamic universes, incomplete information and uncertainty. There are petabytes of data available, with new data becoming available all the time, and the agent itself produces many new data products in the course of fulfilling the user's goal—data products that could be used to fulfill subsequent goals. There is also considerable uncertainty — uncertainty of the time that particular data will be available, or whether the data will arrive at all, uncertainty in the quality of data, even uncertainty as to whether a given processing algorithm will succeed. To cope with this uncertainty, the agent may need to poll for data availability or try alternative courses of action if the one it is pursuing seems unpromising.

We have developed a planner-based agent, called IMAGEbot, to automate data production. The data production problem may be viewed as a planning problem in which the initial state describes the current set of available data products, and whose goal state describes the properties of the desired high-level data products. Planner operators correspond to data transformation and generation tools. IMAGEbot takes data product requests as high-level goals and executes the commands needed to produce the requested data products.

We adopt a planning approach somewhat similar to Graphplan, consisting of a Graphplan-style reachability analysis and a constraint-based search. However, the large universe of the data production problem makes the grounded planning graph of Graphplan inapplicable; instead, we choose a lifted representation where actions and plans contain variables. Because of the lifted representation, and the uncertain and dynamic nature of the data production problem, the reachability analysis and search cannot be separated; instead, IMAGEbot is interleaves planning, constraint reasoning and execution.

In this paper, we report on our work on IMAGEbot, with a focus on the constraint reasoning that underlies planning, sensing and execution. Section 2 gives an overview of the IMAGEbot system architecture and high-level planning approach; Section 3 discusses our constraint-based approach to sensing; Section 4 discusses a novel constraint propagation algorithm based on the planning graph. Section 5 discusses

*QSS Group Inc

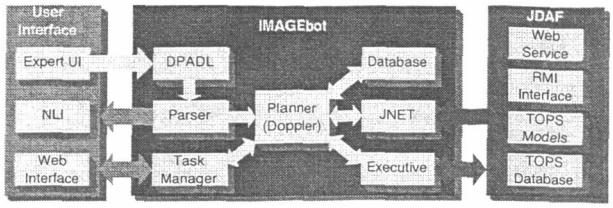

Figure 1: The architecture of IMAGEbot

interleaved planning and execution.

2 IMAGEbot Overview

2.1 System Architecture

The architecture of the IMAGEbot agent is depicted in Figure 1. The main components are:

JDAF: The **J**ava **D**istributed **A**pplication **F**ramework comprises execution environment for IMAGEbot; it provides the agent with a common API for data-processing programs and ecological forecasting models.

DPADL: The **D**ata **P**rocessing **A**ction **D**escription **L**anguage [Golden, 2002] is used to provide action descriptions of data-processing programs and available data sources. Goals, in the forms of data product requests, can also be described in DPADL. To support both fine-grained and flexible sensing, DPADL allows constraints to make calls calls to the underlying runtime environment (Section 3).

DoPPLER: The **D**ata **P**rocessing **Planner** accepts goals in the form of data descriptions and synthesizes and executes data-flow programs. It reduces the planning problem to a CSP whose solution provides a solution to the original planning problem.

JNET: **J**ava Constraint **Net**work is a constraint representation and reasoning framework that provides the agent with constraint propagation and search capabilities.

The architecture provides a planning framework that interleaves planning with constraint reasoning and plan execution.

2.2 Planning Approach

Planning in IMAGEbot is a two-stage process. The first stage consists of a Graphplan-style reachability analysis [Blum & Furst, 1997] to derive heuristic distance estimates for the second stage, a constraint-based search. These stages are not entirely separate, however; constraint propagation occurs even in the the graph-construction stage, and the graph is refined during the constraint-search phase.

Lifted planning graphs

Planning domains are specified in DPADL. From the planning problem specification, the planner incrementally constructs a directed graph, similar to a planning graph [Blum & Furst, 1997], but using a lifted representation (*i.e.*, containing variables). This graph is used to obtain distance estimates for heuristic search, and is also the basis for the construction of the CSP. Arcs in the graph are analogous to causal links [Penberthy & Weld, 1992]. A causal link is triple $\langle \alpha_s, p, \alpha_p \rangle$, recording the decision to use action α_s to support precondition p of action α_p. However, instead of recording a commitment of support, it indicates the *possibility* that α_s supports p. The lifted graph contains multiple ways of supporting p; the choice of the actual supporter becomes a constraint satisfaction problem. We add an extra term to the arc for bookkeeping purposes – the condition, $\gamma_p^{\alpha_s}$, needed in order for α_s to achieve p. A link then becomes $\langle \alpha_s, \gamma_p^{\alpha_s}, p, \alpha_p \rangle$.

Given an unsupported precondition p of action α_p, our first task is to identify all the actions that could support p. Because the universe is large and dynamic, identifying all possible ground actions that could support p would be impractical, so instead we use a lifted representation, identifying all action *schemas* that could provide support. Given an action schema α, we determine whether it supports p by *regressing* p through α. The result of regression is the formula $\gamma_p^{\alpha_s}$. If $\gamma_p^{\alpha_s} = \bot$, then α does not support p. Initial graph construction terminates when all preconditions have support or (more likely) a potential loop is detected.

From planning to constraints

After the graph is constructed, heuristic distance estimates for guiding the search are computed, and a constraint problem representing the search space is incrementally built. It is incremental because the planning graph comprises a compact representation of the search space, in which each action node can represent multiple concrete actions in the final plan. Since the number of possible actions can be large, even infinite, we cannot simply generate all of them at once but do so lazily during search. This is handled using a dynamic CSP (DCSP), in which new variables and constraints can be added for each new action and causal link in the plan.

The CSP contains: 1) boolean variables for all arcs, nodes and conditions; 2) variables for all parameters, input and output variables and function values; 3) for every condition in the graph, a constraint specifying when that condition holds (for conditions supported by arcs, this is just the XOR of the arc variables); 4) for conjunctive and disjunctive expressions, the constraint is the respective conjunction or disjunction of the boolean variables corresponding to appropriate sub-expressions; 5) for every arc in the graph, constraints specifying the conditions under which the supported fluents will be achieved (i.e., $\gamma_p^{\alpha} \Rightarrow p$, where γ_p^{α} is the precondition of α needed to achieve p) ; 6) user-specified constraints; and 7) constraints representing structured objects.

Constraint-based search

After converting the planning problem to a CSP, the planner searches the CSP for a solution. At a high level, the planner, guided by heuristic distance estimates extracted from the planning graph, selects subgoals to achieve and actions to achieve them (Algorithm 2). After the subgoal and action selection, the planner (or more accurately, the CSP solver) finds values for variables representing planner action parameters. This is necessary to make actions executable. During the search, propagation is performed whenever a value is assigned to a variable. The search is an iterative process involving possible backtracks; that is, if there are no valid parameters for a chosen action, the planner has to search for another plan; if it is impossible to extract a plan from the current plan

graph, the planning graph is extended or search fails.

3 Constraint-based sensing

In order to find out what data products relevant to the task at hand are available, the agent needs to sense its environment. One way of doing this is to introduce *sensing actions* [Golden & Weld, 1996], which the agent can execute in order to obtain information. This approach has the advantage that it can be used to capture sensing actions that have preconditions, but it also requires the plan to be at least partially executed before the information can be obtained. We follow an alternative approach of representing low-cost precondition-free sensors using *procedural constraints*. That is, we can implement constraints as procedures that can perform database queries or invoke other information-gathering operations in the course of identifying the domain of values a given variable can have. This constraint-based sensing approach is much more flexible than the sensing-action approach, as the order of sensing operations is based on constraint propagation, and information dependencies are inherently multi-directional. For example, suppose we have a set of satellite images, each of which corresponds to a given region of the Earth's surface for a given day. If we have a specific satellite image, we may invoke methods to determine the region and day for that image. On the other hand, if we know we need an image corresponding to a given region and day, we may query a database to find out what images are available for the time and place in question. The specific set of operations performed depends on which variables are bound (or appropriately restricted), *i.e.*, what information is "known" to the constraint solver. Invoking a sensing operation may trigger further sensing through constraint propagation. For example, suppose we are interested finding a high resolution satellite image of western Oregon for a day in June that had no rainfall. We can perform a database query to find out what images are available over western Oregon for June. Once the images are known, we can query to find the resolution of each one, eliminating from consideration those of insufficient resolution. We can then query to determine the day that each image was captured, then do another query to determine the precipitation for that day. Finally, we remove from consideration all images for days that had non-zero precipitation. The order of sensing operations depends on what information is "known" and what information is needed.

We can also represent more traditional sensing actions, using actions that produce new objects (data files), which contain information. Acquiring these objects can, in turn, trigger more constraint propagation, resulting in more implicit sensing. For example, a data-acquisition action may obtain a set of satellite images from a remote location. Once these images are available, additional operations can be performed to obtain information about the images, such as data quality. These additional operations can be implemented as constraints rather than actions, which removes them from the set of deliberate decisions that the planner needs to make.

4 Action-based Constraint Propagation

As we have discussed, data production problems, due to their large, uncertain and dynamic universes, are not suitable for a grounded representation. The lifted planning graph is a much more concise representation than the grounded planning graph, but it is potentially less informative, which makes conventional constraint propagation and search less effective. The CSP derived from the lifted planning graph contains variables with infinite domains [Golden & Frank, 2002], so there is no way to enumerate solutions by search alone, yet the traditional constraint propagation that establishes certain levels of consistency does not work well either. For example, we have a constraint propagator in JNET that enforces a partial[1] *generalized arc-consistency* (GAC) [Bessiere & Ch, 1997; Katsirelos & Bacchus, 2001]. The definition of GAC is built upon the variables and their values; namely, a CSP is GAC if all its variables are GAC; a variable is GAC is all its values are GAC; a value v of a variable x is GAC if it has support from other variables in every constraint on x. Establishing consistency requires evaluating every value to see if it satisfies certain constraints, which is not possible in general for infinite variable domains. A combination of propagation and search will eventually find a solution, but propagation does not become informative until late in the search .

We have developed a new constraint propagation algorithm that propagates changes among the actions in the planning graph, which yields much more information, even before search begins. It not only restricts the domains of variables by eliminating inconsistent values, but it also may add values to the variable domains when new information is available (e.g., a new object is created). In this section, we first describe the propagation algorithm, then illustrate how it works with an example, and discuss its role in the planning search and constraint search.

4.1 Algorithm

Formally, a data-processing action schema can be seen as a tuple $\langle \mathcal{I}, \mathcal{O}, \mathcal{P}, \Pi, \mathcal{E}, \chi \rangle$, where $\mathcal{I}$,$\mathcal{O}$,$\mathcal{P}$ are the *input* variables, *output* variables and *parameters* respectively. The parameters are unknowns that may appear in constraints on either or both input and output. Π is the *precondition*, $\mathcal{E}$ is *effects* and χ is a procedure for executing the action that may reference any variable in $\mathcal{I} \cup \mathcal{P}$ and must set every variable in $\mathcal{O}$. A lifted planning graph can be seen as a partially ordered set of actions $(A, \prec)$, where $a \prec b$ *iff* action a supports b or a supports c and $c \prec b$. In the CSP derived from the lifted planning graph, we have constraints specifying the relationships among variables inside an action and constraints specifying relationships of two actions if one supports another. For an individual action, if something changes, for example, if a value is assigned to a variable in the action input due to search , the change to this variable can be propagated to other variables in the output, which may change their domains. For two actions a and b, where a supports b, changes in the input of b can be propagated to the output of a; similarly, changes in the output of a can be propagated to the input of b. The idea of this propagation is outlined in Algorithm 1.

In Algorithm 1, function `enforce`(P, C_a) enforces every constraint $c \in C_a$ associated with action a. It restricts

[1] We call it partial GAC for two reasons: 1) not every constraint procedure enforces the GAC; and 2) not every constraint is executed in the propagation.

Algorithm 1 Action Constraint Propagation

Given a lifted plan graph G. Let A be the set of actions in G, let $P = (X, D, C)$ be the CSP derived from the lifted plan graph, and let A' be a subset of actions to be propagated:

propagate(G, A, P, A')

1. **while** $(A' \neq \emptyset)$ **do**
   (a) **let** $a \leftarrow$ `an action removed from` A'
   (b) **let** $C_a \leftarrow$ `constraints relevant to` a
   (c) $< d(\mathcal{I}(a)), d(\mathcal{O}(a)) > \leftarrow$ `enforce`(P, C_a)
   (d) **for** $(\forall i \in \mathcal{I}(a)$ s.t. $d(i) = \emptyset)$
   `remove supporting link to` i
   (e) **for** $(\forall o \in \mathcal{O}(a)$ s.t. $d(o) = \emptyset)$
   `remove supporting link from` o
   (f) **for** $(\forall i \in \mathcal{I}(a)$ s.t. $d(i)$ changed)
      i. **for** $(\forall b \in A$ s.t. b `supports` $a)$
      if (`revise`$(P, \mathcal{O}(b), i))$ $A' \leftarrow A' \cup \{b\}$
   (g) **for** $(\forall o \in \mathcal{O}(a)$ s.t. $d(o)$ `changed`$)$
      i. **for** $(\forall b \in A$ s.t. a `supports` $b)$
      if (`revise`$(P, \mathcal{I}(b), o))$ $A' \leftarrow A' \cup \{b\}$
2. **return**

domains of variables in c by eliminating inconsistent values. Function `revise`$(P, \mathcal{O}(b), i)$ (or `revise`$(P, \mathcal{I}(b), o)$) computes the domains of variables in $\mathcal{O}(b)$ (or $\mathcal{I}(b)$), where i is an input (or o an output) of action a and action b supports (or is supported by) a. The function `revise` may remove inconsistent values or add newly discovered values depending on the planning graph structure. It returns true if any variable domain has been revised, in which case the action b is added to A', waiting to be propagated.

In addition to removing inconsistent values or discovering new values for variables in an action, this propagation also removes certain supporting links if it identifies inconsistency. If all links from an action a supporting other actions are removed, the action a is useless in the planning graph so it can be safely removed. If all links to an input of an action a are removed, this action cannot be executed because one of its inputs does not have support. The planner either has to find other support for this action (e.g., expanding the planning graph by inserting more actions) or remove this action from the planning graph.

4.2 Example

For illustration, we consider a simplified version of constructing a *mosaic*. Many satellites continuously image whatever portion of the Earth they pass over, like giant hand-held scanners. For convenience, the resulting *swath* data is usually reprojected into onto a 2D *map* and chopped up into *tiles*, corresponding to a regular grid drawn over the map. To obtain the data pertaining to a particular region of the Earth, we first identify and obtain the tiles that cover that region and then combine them into a single image, known as a mosaic, and crop away the pixels outside the region of interest.

These tiles are represented in the planner as first-class objects. The attributes of a tile describe, among other things,

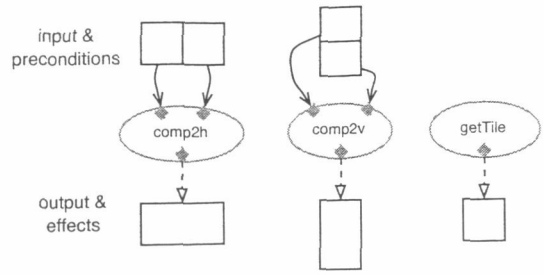

Figure 2: The planner actions: the dots inside actions are inputs and outputs. Parameters are not shown.

the physical measurement the data in the tile represent, the position of the tile on the grid, the projection used to flatten the globe, and the region of the Earth covered by the pixels in the image. For simplicity, we assume in this example that tiles have only two attributes: the *region* a tile covers and the *cloudiness* when the image was taken. A simplified task becomes to take some tiles from thousands of available tiles and compose them to create a mosaic that covers a specified region without too much cloud cover.

Specifically, a *region* is a pair of points $\langle ul, lr \rangle$ where ul is the upper-left corner and lr the lower-right corner. A point is a pair of coordinates (x, y). Normally x and y would be longitude and latitude, but as a further simplification, we will assume both x and y are non-negative integers. The cloudiness is represented by a real number from 0 to 1, where 0 is clear sky and 1 is totally obscured. Further, we assume there are only three actions the planner may take: compose two tiles horizontally (*comp2h*) or vertically (*comp2v*), or get a tile with its *ul* point as a parameter (*getTile*). A real mosaic command is not limited to combining two tiles. Figure 2 shows action preconditions and effects with respect to the region. In addition, the effect of composing two images is that their combined cloudiness is treated as the maximum of the cloudiness of the input tiles.

A problem instance we consider here consists of some small tiles, such as $\langle (0,0), (1,2) \rangle$, or $\langle (2,3), (3,5) \rangle$. The goal is to compose a mosaic for the region $\langle (0,0), (3,2) \rangle$ with no more than 15% cloud cover. This mosaic is composed of tiles $B_1, B_2, ..., B_6$, which may or may not available locally; if not, we assume that action *getTile*$((x, y))$ can be executed to get any available tiles $\langle (x, y), (x + m, y + n) \rangle$.

The planning graph created by the planner is shown in Figure 3, where nodes represent lifted actions and arcs the supporting relations. The dots inside action nodes are inputs and outputs of the actions, each representing a set of objects, possibly infinite. At the time when a CSP is derived from this planning graph, these unknown objects, inputs and outputs of the actions and their parameters, are represented as variables with infinite domains.

The action-based constraint propagation can be invoked to restrict some of the infinite domains. Since the planner goal $\langle (0,0), (3,2) \rangle$ is known, the output of action *comp2v*, which supports the goal, is also known; applying the propagation on *comp2v*, we have the domains

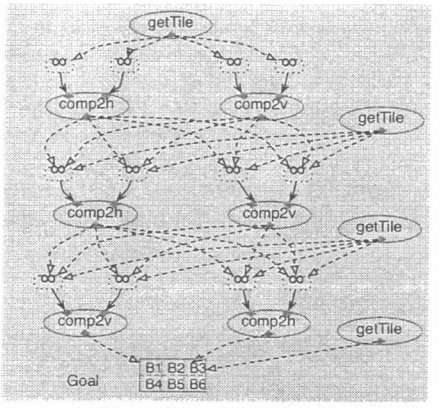

Figure 3: A planning graph

of its two inputs, both of which are singletons, namely $\{\langle(0,0),(3,1)\rangle\}$ and $\{\langle(0,1),(3,2)\rangle\}$. Similarly, the output of *comp2h* is known; applying the propagation on *comp2h*, we have the domains of its two inputs, both of which contain two regions: $\{\langle(0,0),(1,2)\rangle, \langle(0,0),(2,2)\rangle\}$ and $, \{\langle(1,0),(3,2)\rangle, \langle(2,0),(3,2)\rangle\}$, respectively. The changes to inputs of the these actions are propagated to the next level actions supporting them.

When propagation stops, we have a much more limited search space as shown in Figure 4, where the tiles in the inputs and outputs are restricted to specified regions. These tiles crossed out are the ones eliminated by propagation from the initial state. Notice also that many links appearing in Figure 3 have been removed by the propagation. For example, all links from *comp2v* to *comp2v* have been removed.

We also have a goal constraint requiring the cloudiness of the image to be at most 15%. Propagating this constraint backward through the graph results in the requirement that each input image has a cloudiness of at most 15% (not shown). However, the cloudiness of the tiles is unknown at planning time, so no further propagation or pruning can be done until the plan is at least partially executed. We continue with this example in the next section.

5 Planning and Execution

Although our constraint-based approach to sensing helps to cope with large, unknown domains, there is still some uncertainty, even for a "complete" plan. Data products may turn out to be of a lesser quality than expected, due to cloud cover for instance, or may even turn out to be missing entirely. Processing algorithms may fail to perform as well as expected, perhaps due to problems with the input data, or they may simply crash. Some quality problems can be automatically detected, but only after the data products are in hand, meaning after the plan has been at least partially executed. Fortunately, the non-destructive nature of data production domains means the cost of plan execution is limited to the time and resources consumed, so it is natural to view plan execution as an extension of the search process. If partial execution of a plan reveals a violation of a constraint or preference, it is a simple matter to backtrack and try something else, since there are

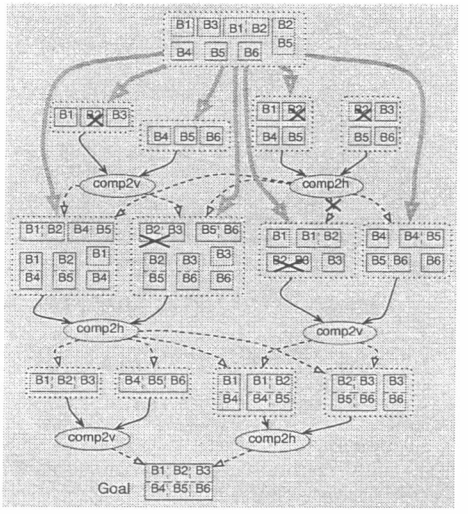

Figure 4: Constraint propagation in the planning graph. Objects in a dotted rectangles are inputs to an action; an object divided by dashed line is a composed object; single objects are available in the initial state or can be obtained with *getTile* (not shown).

Algorithm 2 Plan construction and execution. Iteratively supports subgoals and executes actions until all goals are supported and all actions are executed. The keyword **pick** indicates a choice that is not a backtrack point. The keyword **choose** indicates nondeterministic choice (backtrack point) The keyword **fail** indicates a backtrack.

public void PlanAndExecute(goal, actions)

1. **let** $G \leftarrow$ BuildPlanGraph(goal, actions)
2. **let** $P \leftarrow$BuildConstraintNet(G), $A \leftarrow$ Actions in G
3. **let** agenda $\leftarrow$ {goal}, unexecuted $\leftarrow$ {goal}
4. **set** d(goal) $\leftarrow$ {true}
5. **while** (propagate(G, A, P, A) returns false)
   if (ExpandGraph(G, P) returns false) **fail**
6. **while** (unexecuted $\neq \emptyset$) **pick**
   (a) **pick** $\alpha \in$unexecuted
   if (execute(α) returns true)
   remove α from unexecuted
   (b) **let** $p \leftarrow$ remove from agenda
   i. **choose** $\langle \alpha_s, \gamma_p^{\alpha_s}, p, \alpha_p \rangle$ in G
   ii. add $\gamma_p^{\alpha_s}$ to agenda and set $d(\gamma_p^{\alpha_s}) = \{true\}$
   iii. add α_s to unexecuted
   iv. **if** (propagate(G, A, P, $\{\alpha_s, \alpha_p\}$) returns false) **fail**
   (c) ExpandGraph(G, P)

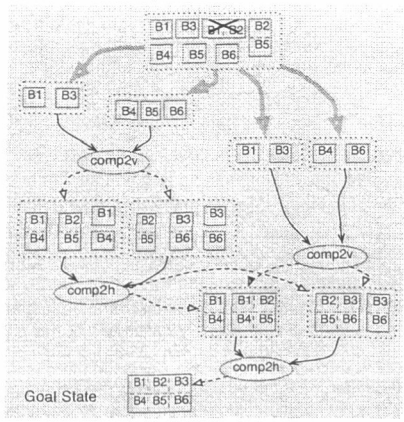

Figure 5: Partial execution provides additional information, which allows additional constraint propagation and pruning of the planning graph.

no state changes to be undone. Furthermore, actions may be executed before the plan is complete, yielding information to reduce search or choose between competing options. For example, if there are two candidate data sets, each of unknown quality and each of which requires different processing steps, the planner can execute the actions to obtain both sets of data and decide which one to use before wasting time planning out all the processing operations for data that may not be used.

Here, again, the planning graph representation is useful, because it provides a guide to which data sources and actions are relevant to a problem without requiring a complete plan to be generated. Once an action has been executed and its outputs produced, the output variables are instantiated with the results from execution and the constraints are re-propagated, which may further restrict the domains of other variables, reducing the amount of search.

5.1 Example

To continue our previous example, suppose that we execute all the getTile actions in the planning graph before doing any explicit search. Since getTile obtains the actual images, constraint propagation will result in determining the cloudiness of each of the images. Recall that the domain for each cloudiness variable was $[0 \ldots 0.15]$, since the maximum allowable cloudiness specified in the goal is 0.15,. During propagation, the actual cloudiness of each tile will be determined and intersected with the original domain of $[0 \ldots 0.15]$. If the value is greater than 0.15, the domain will become empty. Suppose the tile spanning B1 and B2 has cloudiness of 0.25, and all the others have cloudiness of 0.0. This result is propagated through the action graph, eliminating a number of values and two actions (Figure 5).

6 Conclusions

IMAGEbot is implemented and has been integrated into an ecological forecasting application [Golden *et al.*, 2003], which produces "nowcasts" and forecasts of socioeconomic importance, such as crop health and fire risk.

We believe the constraint-based sensing and planning-graph propagation approaches introduced in this paper would be equally suitable to other software domains that involve large, unknown dynamic domains. Related applications to which planners have been applied include Internet softbots [Golden, 1998; Etzioni, Golden, & Weld, 1997], web services [Srivastava & Kholer, 2003], image processing [Lansky, 1998; Chien *et al.*, 1997], and grid-based computing [Blythe *et al.*, 2003].

References

[Bessiere & Ch, 1997] Bessiere, C., and Ch, J. 1997. Arc-consistency for general constraint networks: Preliminary results. In *Proceedings of IJCAI-97*, 398–404.

[Blum & Furst, 1997] Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *J. Artificial Intelligence* 90(1–2):281–300.

[Blythe *et al.*, 2003] Blythe, J.; Deelman, E.; Gil, Y.; Kesselman, C.; Agarwal, A.; Mehta, G.; and Vahi, K. 2003. The role of planning in grid computing. In *Proc. 13th Intl. Conf. on Automated Planning and Scheduling (ICAPS)*.

[Chien *et al.*, 1997] Chien, S.; Fisher, F.; Lo, E.; Mortensen, H.; and Greeley, R. 1997. Using artificial intelligence planning to automate science data analysis for large image database. In *Proc. 1997 Conference on Knowledge Discovery and Data Mining*.

[Etzioni, Golden, & Weld, 1997] Etzioni, O.; Golden, K.; and Weld, D. 1997. Sound and efficient closed-world reasoning for planning. *J. Artificial Intelligence* 89(1–2):113–148.

[Golden & Frank, 2002] Golden, K., and Frank, J. 2002. Universal quantification in a constraint-based planner. In *Proc. 6th Intl. Conf. Automated Planning Systems*.

[Golden & Weld, 1996] Golden, K., and Weld, D. 1996. Representing sensing actions: The middle ground revisited. In *Proc. 5th Int. Conf. Principles of Knowledge Representation and Reasoning*, 174–185.

[Golden *et al.*, 2003] Golden, K.; Pang, W.; Nemani, R.; and Votava, P. 2003. Automating the processing of earth observation data. In *International Symposium on Artificial Intelligence, Robotics and Automation for Space*.

[Golden, 1998] Golden, K. 1998. Leap before you look: Information gathering in the PUCCINI planner. In *Proc. 4th Intl. Conf. AI Planning Systems*.

[Golden, 2002] Golden, K. 2002. DPADL: An action language for data processing domains. In *Proceedings of the 3rd NASA Intl. Planning and Scheduling workshop*, 28–33. to appear.

[Katsirelos & Bacchus, 2001] Katsirelos, G., and Bacchus, F. 2001. GAC on conjunctions of constraints. In *Proceedings of CP-2001*.

[Lansky, 1998] Lansky, A. 1998. Localized planning with action-based constraints. *Artificial Intelligence* 98(1–2):49–136.

[Penberthy & Weld, 1992] Penberthy, J., and Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. 3rd Int. Conf. Principles of Knowledge Representation and Reasoning*, 103–114.

[Srivastava & Kholer, 2003] Srivastava, B., and Kholer, J. 2003. Web service composition - current solutions and open problems. In *ICAPS 2003 Workshop on Planning for Web Services*. available at http://www.isi.edu/info-agents/workshops/icaps2003-p4ws/program.html.